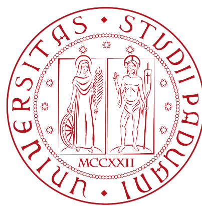

Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Triennale in

Ingegneria Informatica

# Database NoSql analisi prestazionale di Redis

Relatore: Prof. Giorgio Maria Di Nunzio

Laureando: Gabriele Casagrande

Anno Accademico 2013/2014

# Indice

**1 Introduzione** **1**
1.1 NOSQL . . . 1
1.2 Scalabilità e il teorema CAP . . . 2
1.3 Classificazione dei database NoSQL . . . 3

**2 Redis** **5**
2.1 Introduzione . . . 5
2.2 Persistenza dei dati . . . 5
2.2.1 RDB - Redis DataBase . . . 6
2.2.2 AOF - Append Only File . . . 6
2.3 Il motore di Redis . . . 7
2.4 Sicurezza . . . 7
2.4.1 Autenticazione . . . 8
2.4.2 Disabilitazione comandi . . . 8
2.4.3 "NoSQL-injection" . . . 8
2.4.4 La cifratura dei dati . . . 8
2.5 Transazioni . . . 8
2.6 Replicazione . . . 9
2.7 Strutture dati di Redis . . . 9
2.8 Librerie software . . . 14

**3 Progettazione "clone di Twitter"** **17**
3.1 Progettazione con Redis . . . 17
3.1.1 Utenti . . . 17
3.1.2 Post . . . 18
3.2 Database Twitter con MySql . . . 19
3.2.1 Raccolta e analisi requisiti . . . 19
3.2.2 Progettazione concettuale . . . 19
3.2.3 Progettazione logica . . . 20
3.2.4 Progettazione fisica . . . 20

**4 Analisi prestazionale** **23**
4.1 Inserimenti . . . 23
4.1.1 Inserimento in Redis . . . 23
4.1.1.1 Python . . . 24
4.1.1.2 Python Pipeline . . . 24
4.1.1.3 File txt . . . 24
4.1.2 Inserimento in MySql . . . 25
4.1.2.1 Python . . . 25
4.1.2.2 Python bulk . . . 26
4.1.2.3 txt . . . 26
4.1.3 Analisi . . . 27
4.2 Query . . . 28
4.3 Ristrutturazione database Redis . . . 32

**5 Conclusioni** **39**

# Elenco delle figure

1.1 Teorema CAP . . . . . . . . . . 3
1.2 Confronto NoSQL - Dimensioni Vs Complessità . . . . . . . . . . 4

3.1 Schema ER clone twitter . . . . . . . . . . 19
3.2 Schema Logico clone twitter . . . . . . . . . . 20

4.1 Grafico inserimenti pipeline . . . . . . . . . . 28
4.2 Grafico prestazioni query Redis Vs MySql . . . . . . . . . . 30
4.3 Grafico utilizzo memoria Redis Vs MySql . . . . . . . . . . 32
4.4 Grafico prestazioni query Redis Lua Vs Redis Lua 1Zset . . . . . . . . . . 34
4.5 Grafico utilizzo memoria Redis Lua Vs Redis Lua 1Zset . . . . . . . . . . 35
4.6 Grafico prestazioni query Redis con 10 followers . . . . . . . . . . 36
4.7 Grafico prestazioni query Redis con numero di followers random . . . . . . . . . . 37

# Elenco delle tabelle

2.1 Librerie con supporto a Redis . . . . . . 15

4.1 Benchmark inserimenti Redis Python . . . . . . 24
4.2 Benchmark inserimenti Redis Python Pipeline . . . . . . 24
4.3 Benchmark inserimenti Redis file . . . . . . 25
4.4 Benchmark inserimenti MySql Python . . . . . . 26
4.5 Benchmark inserimenti MySql Python Pipeline . . . . . . 26
4.6 Benchmark inserimenti singoli MySql File . . . . . . 27
4.7 Benchmark inserimenti bulk 500 MySql File . . . . . . 27
4.8 Confronto Redis Vs MySql query costruzione time-line . . . . . . 29
4.9 Benchmark query Redis con script Lua costruzione time-line . . . . . . 31
4.10 Confronto utilizzo memoria Redis Vs MySql . . . . . . 31
4.11 Benchmark Redis reimplementazione con Zset . . . . . . 33
4.12 Utilizzo memoria Redis reimplementazione con Zset . . . . . . 34
4.13 Benchmark Redis Zset - Redis List - MySql query con "pochi" followers . . . . . . 36
4.14 Benchmark Redis Zset - Redis List - MySql query con "tanti" followers . . . . . . 37
4.15 Benchmark Redis Zset - Redis List - MySql query con "tanti" followers e 100 post/utente . . . . . . 38
4.16 Benchmark Redis Zset - Redis List - MySql query con "tanti" followers e 200 post/utente . . . . . . 38

# Capitolo 1

# Introduzione

Durante questi ultimi anni, le tecniche e gli strumenti usati per la persistenza e la ricerca dei dati hanno avuto una forte crescita. Anche se, quasi sicuramente, non saranno mai abbandonati i database relazionali, possiamo sicuramente dire che il panorama attorno alle basi di dati è cambiato radicalmente ed è in continua evoluzione.

Tra tutti questi nuovi strumenti spicca Redis, un database non relazione molto interessante sotto molti punti di vista: alte prestazioni, facilità di apprendimento e adattabile alle esigenze del programmatore.

In questa tesi verranno studiate le sue caratteristiche, funzionalità, pregi e difetti basandosi sul libro di Josiah L. Carlson 2013 "Redis in action" Manning e traendone esempi di utilizzo pratico. Verranno fatti alcuni benchmark iniziali per capire le potenzialità di Redis e prendere familiarità con le strutture dati che mette a disposizione, verrà poi preso come caso di studio la realizzazione di un clone di Twitter utilizzando Python e Redis, e successivamente la sua reimplementazione con MySql.

## 1.1 NOSQL

NoSQL è l'acronimo di "Not only SQL" ed è usato per identificare tutti quei database che non fanno uso di un modello di dati relazionale, difatti concetti come tabella, indici, chiavi primarie o esterne non è detto che siano presenti.
Il movimento NoSQL nasce nel 2009 ed è in rapida crescita, sono infatti già molte le aziende, dalle piccole realtà alle grandi multinazionali che decidono di passare a questo tipo di database.
Il motivo di principale di questi cambiamenti è che permette alla base dati e di conseguenza alle applicazione corrispondenti di scalare orizzontalmente superando egregiamente le limitazioni dei classici database relazionali.
Tra i pionieri troviamo Google con il suo BigTable e Amazon che ha dato vita a DynamoDB, entrambi i progetti proprietari hanno dimostrato la necessita di uscire dai classici schemi relazioni per permettere un'efficiente scalabilità dei loro servizi.
Da questi esperimenti sono nati i primi database NoSql, quasi tutti Open Source, ognuno caratterizzato da strutture dati diverse e altamente ottimizzate, ma che essenzialmente hanno in comune queste caratteristiche principali:

- schema-free
- la gestione di un'enorme quantità di dati garantendone comunque un rapido accesso

- la facilità di replicazione
- l'utilizzo di api per l'interfacciamento con i più noti linguaggi di sviluppo ed un abbandono di SQL

## 1.2 Scalabilità e il teorema CAP

All'inizio del XXI secolo, prima ancora della nascita dei database NoSQL, c'è stato un aumento esponenziale di utenti che iniziavano a consumare enormi quantità di informazioni collegandosi tramite internet a basi di dati sempre più grandi. Scalare verticalmente stava diventando troppo costoso e insufficiente a mantenere la disponibilità dei servizi accettabile; si è dovuto quindi iniziare a scalare orizzontalmente per bilanciare il continuo carico di richieste, ma portando con se non pochi problemi.
Nel 2000 Eric Brewer presenta il teorema CAP, il quale afferma l'impossibilità per un sistema informatico distribuito di fornire simultaneamente tutte e tre le seguenti garanzie:

1. **Consistency**: Dopo ogni operazione il sistema si trova in uno stato consistente: tutti i nodi vedono gli stessi dati nello stesso momento.
2. **Availability**: L'eventuale fallimento di un nodo non blocca il sistema.
3. **Partition tolerance**: Il sistema continua a operare anche in seguito ad un partizionamento della rete.

A seconda se il sistema decida di sacrificare Availability o Consistency o Partition tolerance è possibile classificare i database distribuiti in tre categorie: CA, CP e AP. Risulta evidente che i tradizionali RDBMS possono essere classificati come sistemi CA, cioè garantiscono consistenza e disponibilità ma non tolleranza alle perdite di messaggi.

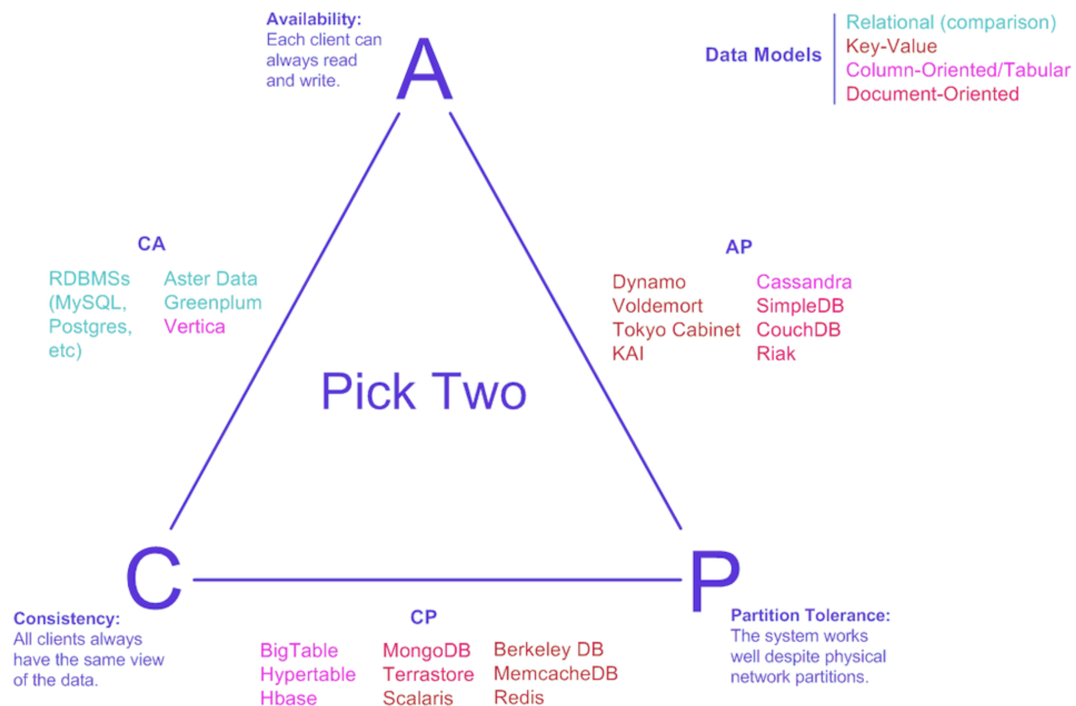


Figura 1.1: Teorema CAP

## 1.3 Classificazione dei database NoSQL

Tra i vari database non relazioni si possono distinguere principalmente 4 grandi gruppi:

1. **Key/Value**: Sono definiti da un semplice dizionario/mappa che permette all'utente di recuperare e aggiornare il valore memorizzato conoscendo la sua chiave

2. **Document**: Memorizza le informazioni come collezioni di documenti. Un documento può contenere informazioni annidate ed ha un formato riconosciuto (JSON, XML, etc.) che permette poi al server di eseguire delle query sui dati

3. **Graph**: Rappresentano perfettamente una realtà composta da una fitta rete di connessioni e la modellano sotto forma di nodi e rami di un grafo. Ai nodi come ai singoli rami vengono associate le informazioni attraverso Key-Value store. Se togliamo le relazioni (i rami) assomigliano a tutti gli effetti ad un database documentale.

4. **Column-oriented**: Sistemi che utilizzano ancora le tabelle ma che non fanno alcun tipo di join, le informazioni non sono memorizzate per riga bensì per colonna

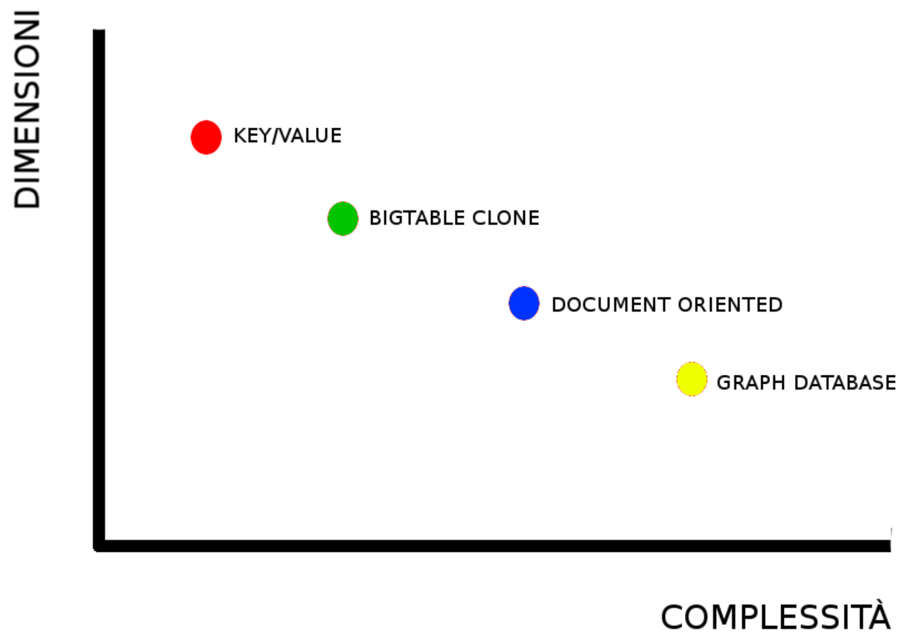


Figura 1.2: Confronto NoSQL - Dimensioni Vs Complessità

I database key/value, trattandosi di semplici associazioni chiave-valore, sono il tipo più semplice! Questa semplicità si paga con un maggior numero di dati da inserire per descrivere le nostre relazioni.

# Capitolo 2

# Redis

## 2.1 Introduzione

Redis [1] è un database non relazione nato nel 2009 per mano di Salvatore Sanfilippo, inizialmente sponsorizzato da vmware ora è supportato da Pivotal.

È un NoSql di tipo Key/Value ma ha due caratteristiche che lo rendono molto diverso dagli altri database della sua stessa categoria: la prima è che lavora completamente in RAM; la seconda è che oltre a fornire il classico salvataggio delle informazioni tramite coppie chiave-valore offre ben altre quattro strutture dati: liste, insiemi, insiemi ordinati e hash.

## 2.2 Persistenza dei dati

Nell'introduzione abbiamo detto che Redis lavora completamente in RAM, questo vuol dire che le operazioni più importanti riguardanti la nostra base di dati vengono gestite e svolte in RAM. In prima lettura questo potrebbe sembrare un clone di Memcached[2], ma non è così infatti Redis utilizza la memoria secondaria per offrire persistenza dei dati:

- La persistenza RDB (Redis DataBase) esegue degli snapshot ad intervalli di tempo specificati.
- La persistenza AOF (Append Only File) invece registra ogni singola operazione di scrittura ricevuto dal server, e sarà quindi in grado di ricostruire l'insieme di dati originale. I comandi vengono registrati utilizzando lo stesso formato come il protocollo Redis stesso. La gestione e l'ottimizzazione del file di log viene fatta completamente in background.
- Se lo si desidera, è possibile disattivare entrambi i meccanismi di persistenza dei dati, in modo tale che i dati esistano fintanto che il server è in esecuzione.
- È possibile combinare AOF e RDB nella stessa istanza. In questo caso, quando viene riavviato Redis, il file AOF verrà utilizzato per ricostruire il database originale.

Analizziamo i due sistemi per capirne pregi e difetti.

[1] http://download.redis.io/redis-stable.tar.gz

[2] sistema cache in Ram per velocizzare le normali operazioni di un database relazione creando delle associazioni chiave/valore

### 2.2.1 RDB - Redis DataBase

RDB è un vero è proprio snapshot, riesce a rappresentare l'immagine di un'istanza del database con un singolo file su disco molto compatto. Questi File sono perfetti per i backup, si può ad esempio pianificare facilmente con Cron un sistema di accumulo di snapshot successivi ad intervalli regolari consentendo di ripristinare facilmente le diverse versioni del database in caso di disastri. RDB massimizza prestazioni di Redis dato che la creazione degli snapshot è affidata ad un fork e garantendo che il server non eseguirà mai operazioni di I/O. RDB, rispetto ad AOF, consente riavvii più rapidi con grandi set di dati.

RDB non è la miglior soluzione se abbiamo bisogno di ridurre al minimo le possibilità di perdita dei dati, come ad esempio un'accidentale interruzione di corrente. Tuttavia è possibile dichiarare degli intervalli di tempo nei quali il server Redis creerà gli snapshot (per esempio dopo che siano passati almeno 5 minuti e 100 scritture). Anche se questa tecnica utilizza una fork() per il salvataggio su disco, se la base di dati è molto grande e la CPU non molto performante si potrebbe avere un'interruzione del servizio di alcune frazioni di secondo.

### 2.2.2 AOF - Append Only File

Questa tecnica si basa sul principio di scrittura di un file di log che contiene tutti i comandi inviati al server. Utilizzando AOF rendiamo Redis molto più robusto, possiamo decidere che le scritture sulla memoria secondaria siano asincrone, che avvengano ogni secondo oppure che siano sincronizzate ad ogni query. Di default AOF si sincronizza ogni secondo con il file su disco e per farlo utilizza un thread.

Il file di log di AOF, non incorrerà mai in problemi di corruzione perchè lavora solo ed esclusivamente scrivendo in coda al file e non vengono mai eseguite operazioni di seek. Nel caso sfortunato in cui un comando Redis non viene completato, al riavvio del servizio il tool redis-check-aof permetterà di completare le operazioni in sospeso. Dato che AOF non è altro che un log di tutte le operazioni eseguite dal server di Redis, lo possiamo utilizzare ad esempio per ripristinare un intero database svuotato dal comando FLUSHALL (l'equivalente del comando SQL DROP database), basterà aprire il file di AOF con un editor di testo, rimuovere il comando sbagliato e dare in pasto a Redis il file appena modificato. Nel momento in cui il file diventa troppo grande, un processo in background inizia a crearne uno nuovo. La scrittura del nuovo file è completamente sicura dato che finchè non ha finito la sua scrittura Redis continua a loggare tutti i comandi sul vecchio file.

Si tratta di un log quindi il file di AOF risulta sempre più grande di un file RDB a parità di dati inseriti. Se si adotta AOF non sincronizzato otterremo le stesse prestazioni di RDB, la sincronizzazione al secondo non risulta pesante e le differenze sono quasi impercettibili, ma di sicuro se vogliamo una garanzia di persistenza dei dati eccellente, scrittura sincrona ogni operazione, di sicuro la velocità del disco fisso diventerà il collo di bottiglia del nostro sistema.

Se si vuole avere un grado di sicurezza paragonabile a quella dei più comuni database relazioni si dovrebbero utilizzare entrambi i metodi di persistenza. Se vogliamo persistenza dei dati, ma possiamo fare a meno di alcuni "minuti di dati" in caso di disastri, si può benissimo utilizzare solo RDB. C'è la possibilità di affidarsi solo ad AOF, ma non è una bellissima idea, primo perchè con RDB possiamo fare dei backup molto snelli, secondo e più importante che con il supporto di RDB abbiamo dei riavvii più rapidi. Al momento di default è attivo solo RDB ma il team di sviluppo di Redis ha in piano un modello di persistenza dei dati che riesca ad unificare entrambe le strategie.

## 2.3 Il motore di Redis

Redis non è altro che un server TCP che utilizza il modello client-server ed implementa quello che è chiamato un protocollo di tipo Request/Response o per semplicità telnet-like. Ciò significa che normalmente una richiesta viene eseguita mediante la seguente procedura:

1. Il client invia una query al server (comando) ed attende sul socket una risposta.

2. Il server elabora il comando e invia la risposta al client.

Vediamo come esempio l'invio del comando PING

```
$ redis-cli
redis 127.0.0.1:6379> ping
PONG
```

Attualmente ci sono quasi 150 comandi [3] disponibili. In questa tesina trattare tutti questi comandi non avrebbe molto senso, mano a mano ne incontreremo di nuovi ne spiegheremo caratteristiche ed utilizzo. Più avanti analizzeremo le differenti strutture dati e per ognuna vedremo alcuni tra i comandi più utilizzati.

Client e server sono collegati tramite un socket di rete, tale collegamento può essere molto veloce (una interfaccia di loopback) o molto lento (una connessione Internet con molti nodi di passaggio).

C'è quindi da considerare il tempo che serve ai pacchetti per viaggiare dal client al server e tornare indietro con la risposta, questo tempo si chiama RTT (Round Trip Time) ed è facile notare che questo influisce drasticamente sulle prestazioni quando il client deve inviare molte richieste al server.

Se ad esempio la connessione con il server Redis viaggi su un collegamento internet molto lento (RTT 250 ms), anche se il server è in grado di elaborare più di 100000 richieste al secondo, non riusciremo mai a servire più di 4 richieste al secondo.

È stato quindi introdotto il concetto di pipeline, altro non è che la possibilità di inviare più comandi uno di seguito all'altro senza attendere le singole risposte che verranno inviate tutte assieme alla fine dell'esecuzione della pipe.

A partire dalla versione 2.6.0 di Redis è stato introdotto il comando EVAL, utilizzato per valutare script Lua [4] grazie all'interprete integrato nel motore di Redis. Scopo principale di questa funzionalità è quello di massimizzare al massimo le performance in query che alternano operazioni di lettura e scrittura, non potendo quindi beneficiare delle ottimizzazioni delle pipeline.

## 2.4 Sicurezza

In questo sezione vogliamo trattare gli aspetti che di Redis riguardanti la sicurezza come il controllo degli accessi o problemi legati a tentativi di attacco dall'esterno. All'inizio Redis è stato progettato per essere accessibile da parte di client affidabili all'interno di ambienti altrettanto affidabili, difatti non è ottimizzato per avere la massima sicurezza, bensì per ottenere le massime prestazioni e semplicità di accesso ai dati. In linea generale se si deve offrire un'istanza di Redis accessibile tramite Internet è bene costruire uno strato software che sarà l'unica entità capace di instaurare delle connessioni al database.

[3] http://redis.io/commands
[4] http://www.lua.org/

### 2.4.1 Autenticazione

Redis offre un rudimentale sistema di controllo di accesso fornendo un semplice sistema di autenticazione che viene attivato agendo sul file di configurazione [5]. Se è attivata l'autenticazione Redis non eseguirà nessun comando inviato dai nuovi client fintanto che questi non invieranno, tramite il comando AUTH, la password di autenticazione. Questa forma di sicurezza è stata pensata come una ridondanza nel caso in cui il sottostrato software presenti falle di sicurezza, anche perchè trattandosi di un normale comando Redis, la password viene inviata in chiaro e quindi soggetta ad attacchi di tipo MITM o semplice sniffing.

### 2.4.2 Disabilitazione comandi

In Redis è possibile disabilitare comandi per offrirne solo un sottoinsieme utilizzabile ai client o addirittura rinominarli in un nome difficile da indovinare. Qui sotto riportiamo l'esempio della rinominazione e disabilitazione del comando DEL.

```
rename-command DEL ""
rename-command DEL b840fc02d524045429941cc15f59e41cb7be6c52
```

### 2.4.3 "NoSQL-injection"

Il protocollo di Redis non ha il concetto di carattere di escape, il protocollo usa stringhe di lunghezza e parametri prefissati. L'unico caso in cui si potrebbe incorrere in attacchi simili all'sql-injection è quando il programmatore permette l'esecuzione di script Lua costruiti prendendo parametri dall'esterno.

### 2.4.4 La cifratura dei dati

Redis non offre nessun supporto di cifratura dei dati. Una possibile soluzione potrebbe essere quella di cifrare ogni singola chiave/valore con l'utilizzo di librerie software esterne a discapito di prestazioni in lettura/scrittura. Altra possibilità potrebbe essere quella di criptare solamente i file su disco adottando un file-system criptato lato hardware/software.

## 2.5 Transazioni

MULTI, EXEC, DISCARD e WATCH sono i 4 comandi con i quali Redis offre il supporto alle transazioni permettendo l'esecuzione si gruppi di comandi con 2 importanti garanzie:

1. Tutti i comandi in una transazione vengono eseguiti in sequenza. Non verrà mai interrotta la sequenza delle operazioni riguardanti una transazione anche se ci sono altri client che inviano comandi in concorrenza.

2. Una transazione è atomica, o vengono eseguite tutte le operazioni o nessuna.

Con il comando MULTI si dichiara al server che tutti i comandi seguenti faranno parte della transazione e verranno quindi accodati, all'esecuzione del comando EXEC Redis avvierà l'esecuzione della transazione.

[5] /etc/redis/redis.conf

```
> MULTI
OK
> INCR foo
QUEUED
> INCR bar
QUEUED
> EXEC
1) (integer) 1
2) (integer) 1
```

È possibile interrompere il processo di accodamento con il comando DISCARD.

Nel caso ci siano un errore (ad esempio di sintassi) prima dell'esecuzione del comando EXEC la transazione verrà automaticamente scartata, nel caso invece che l'errore si verifichi dopo tale invocazione Redis continuerà ad eseguire tutti i comandi della coda segnalando l'errore riscontrato. Non esistono sistemi di roll-back che permettano in caso di errore di ritornare allo stato precedente.

## 2.6 Replicazione

Come tutti i database NoSQL anche Redis offre un servizio di replicazione su più nodi. Si basa sul concetto di master-slave. Redis implementa una replicazione asincrona dove il nodo master può avere più nodi slave, anche quest'ultimi accettano connessioni dai loro pari. La rete può quindi risultare simile ad un grafo. La replicazione è non-bloccante sia lato master (il master serve i client anche quando sta sincronizzando gli slave) che lato slave (serve i client anche se la connessione al master cade) a meno per quest'ultima non venga specificato diversamente sul file di configurazione.

La replicazione è utile sia come strumento di ridondanza che per scalare orizzontalmente facendo eseguire query pesanti come ad esempio SORT ai nodi slave e riducendo così il carico di lavoro del master.

Il processo di sincronizzazione tra master e slave avviene tramite l'invio di tutti i comandi che permettono di ricreare l'intero data-set del nodo master. Se un nodo si disconnette cerca in automatico di riconnettersi alla rete di replicazione e di farsi mandare la coda di comandi che servono a tornare sincronizzato.

Di default un nodo slave è di sola lettura e quindi rifiuta qualsiasi comando riguardante una scrittura sui dati.

## 2.7 Strutture dati di Redis

Le 5 strutture dati offerte da Redis sono le seguenti:

1. **String**: le stringhe Redis (string) sono la struttura dati più semplice. Quando si pensa a una coppia chiave-valore, ci si riferisce nel mondo Redis alle stringhe, ma il nome non deve trarre in inganno difatti con esse è possibile archiviare tre tipi diversi di dato:
    - Byte string
    - integer
    - floating-point

I comandi base per gestire le stringhe sono principalmente tre: GET SET e DEL che rispettivamente leggono, scrivono ed eliminano una chiave/valore

```
$ redis-cli
redis 127.0.0.1:6379>set hello world
OK
redis 127.0.0.1:6379>get hello
"world"
redis 127.0.0.1:6379>del hello
(integer) 1
redis 127.0.0.1:6379>get hello
(nil)
redis 127.0.0.1:6379>
```

Nella versione 2.6 di Redis sono presenti ben 22 comandi che riguardano le String, tra i più usati ci sono senza dubbio DECR e INCR che in una sola operazione decrementano o incrementano un valore numerico di una chiave, altri sono davvero trovano impiego solo in casi rari e specifici come ad esempio SETBIT che permette di settare a 1/0 il bit di una string indicandone l'offset.

Il caso più comune per le stringhe è memorizzare oggetti (complessi o meno) e contatori. Inoltre, dato che ottenere un valore per chiave è così efficiente, sono spesso usate come cache dati.

2. **Hashes**: la presenza del tipo di dato Hash in Redis è già un valido motivo per non definirlo un puro database chiave/valore. Per molti versi gli hash sono come le stringhe. La differenza fondamentale sta nel fatto che forniscono un ulteriore livello di indirezione tramite un campo (field). Di conseguenza i corrispondenti di set e get per un hash sono HGET e HSET.

   Rispetto alle semplici stringhe, gli hash forniscono delle possibilità in più. Invece di salvare i dati di utente come un singolo valore serializzato (utilizzando ad esempio JSON), si possono in questo modo usare gli hash per avere una rappresentazione più strutturata e accurata. Si ha così il vantaggio di ottenere, aggiornare e cancellare specifiche porzioni di un dato complesso, senza dover estrarre o scrivere l'intero valore con un aumento delle prestazioni.

```
#creazione della chiave hash-key con una
#sottochiave sub-key il cui valore è value

redis 127.0.0.1:6379>hset hash-key sub-key1 value1
(integer) 1
redis 127.0.0.1:6379>hset hash-key sub-key2 value2
(integer) 1
redis 127.0.0.1:6379>hset hash-key sub-key1 value1
(integer) 0

#estrazione di tutte le sottochiavi di hask-key

redis 127.0.0.1:6379>hgetall hash-key
1) "sub-key1"
2) "value1"
3) "sub-key2"
```

```
4) "value2"

#eliminazione di una sottochiave di hask-key

redis 127.0.0.1:6379>hdel hash-key sub-key2
(integer) 1
redis 127.0.0.1:6379>hdel hash-key sub-key2
(integer) 0
redis 127.0.0.1:6379>hget hash-key sub-key1
"value1"

#stampa di tutti i valori delle sottochiavi di hash-key

redis 127.0.0.1:6379>hgetall hash-key
1) "sub-key1"
2) "value1"
```

3. **List**: Le liste Redis permettono di organizzare collezioni di elementi omogenei ovviamente associati a una data chiave. E' possibile aggiungere/rimuovere valori in testa/coda ad una lista, o manipolare valori a un dato indice. Caratteristica fondamentale è che le liste mantengono l'ordine di inserimento, possiamo dunque utilizzarle proprio come degli stack o delle code.

```
#inserimento in coda

redis 127.0.0.1:6379> rpush list-key item
(integer) 1
redis 127.0.0.1:6379> rpush list-key item2
(integer) 2
redis 127.0.0.1:6379>rpush list-key item
(integer) 3

#estrazione di tutti i valori a partire dall'indice 0

redis 127.0.0.1:6379>lrange list-key 0 -1
1) "item"
2) "item2"
3) "item"

#estrazione dell'indice 1

redis 127.0.0.1:6379>lindex list-key 1
"item2"
redis 127.0.0.1:6379>lpop list-key
"item"
redis 127.0.0.1:6379>lrange list-key 0 -1
1) "item2"
2) "item"
```

```
redis 127.0.0.1:6379>
```

4. **Set**: Gli insiemi Redis servono a memorizzare valori univoci. Oltre a permettere di sapere se un dato valore fa parte o meno di un dato insieme, essi sorniscono svariate operazioni di tipo insiemistico come ad esempio l'unione, l'intersezione o la differenza. I set non sono strutture ordinate ma possono comunque risultare utili in molte occasioni, un esempio potrebbe essere l'insieme dei tag associati ad una foto.

```
#aggiunta di elementi all'insieme

redis 127.0.0.1:6379>sadd set-key item
(integer) 1
redis 127.0.0.1:6379>sadd set-key item2
(integer) 1
redis 127.0.0.1:6379>sadd set-key item3
(integer) 1
redis 127.0.0.1:6379>sadd set-key item
(integer) 0

#estrazione degli elementi del set

redis 127.0.0.1:6379>smembers set-key
1) "item"
2) "item2"
3) "item3"

#verifica se l'elemento appartiene all'insieme

redis 127.0.0.1:6379>sismember set-key item4
(integer) 0
redis 127.0.0.1:6379>sismember set-key item
(integer) 1
redis 127.0.0.1:6379>srem set-key item2
(integer) 1
redis 127.0.0.1:6379>srem set-key item2
(integer) 0
redis 127.0.0.1:6379>smembers set-key
1) "item"
2) "item3"
redis 127.0.0.1:6379>
```

5. **ZSet**: L'ultima struttura dati che andiamo ad analizzare è l'insieme ordinato (sorted set), analoghi agli insiemi ad ogni valore è associato un attributo di ordinamento. In un primo momento potrebbe sembrare di scarso utilizzo, invece è tra le strutture dati di Redis più apprezzate ed utilizzate. È ottimo se si vogliono gestire delle graduatorie o molto più banalmente delle liste ordinate per un dato valore come ad esempio la lista ordinata degli articoli di un blog ordinati per data o per numero di commenti.

```
#inserimenti

redis 127.0.0.1:6379>zadd zset-key 728 member1
(integer) 1
redis 127.0.0.1:6379>zadd zset-key 982 member0
(integer) 1
redis 127.0.0.1:6379>zadd zset-key 982 member0
(integer) 0
redis 127.0.0.1:6379>zrange zset-key 0 -1 withscores
1) "member1"
2) "728"
3) "member0"
4) "982"

#estrazione dei valori con score compreso tra 0 e 800

redis 127.0.0.1:6379>zrangebyscore zset-key 0 800 withscores
1) "member1"
2) "728"
redis 127.0.0.1:6379>zrem zset-key member1
(integer) 1
redis 127.0.0.1:6379>zrem zset-key member1
(integer) 0
redis 127.0.0.1:6379>zrange zset-key 0 -1 withscores
1) "member0"
2) "982"
```

## 2.8 Librerie software

Per Redis sono state scritte decine di librerie per svariati linguaggi di programmazione

| | |
|---|---|
| ActionScript | as3redis |
| C | hiredis, credis, libredis |
| C# | ServiceStack.Redis, Booksleeve, Sider, TeamDev Redis Client, redis-sharp, csredis |
| C++ | C++ Client |
| Clojure | carmine, aleph |
| Common Lisp | CL-Redis |
| D | Tiny Redis |
| Dart | DartRedisClient |
| emacs lisp | eredis |
| Erlang | Erldis, Eredis, sharded_eredis, Tideland Erlang/OTP, Redis Client |
| Fancy | redis.fy |
| Go | Go-Redis, Radix, Redigo, Tideland CGL Redis, godis, gosexy/redis, redis.go |
| Haskell | hedis, haskell-redis |
| haXe | hxneko-redis |
| Io | iodis |
| Java | Jedis, JRedis, JDBC-Redis, RJC, redis-protocol, lettuce |
| Lua | redis-lua, lua-hiredis |
| Node.js | node_redis, then-redis, redis-node-client |
| Objective-C | ObjCHiredis |
| Perl | Redis, RedisDB, Redis::hiredis, AnyEvent::Redis, AnyEvent::Redis::RipeRedis, AnyEvent::Hiredis, MojoX::Redis, Danga::Socket::Redis |
| PHP | Predis, phpredis, Rediska, RedisServer, Redisent, Credis |
| Pure Data | Puredis |
| Python | redis-py, txredis, desir, brukva |
| Ruby | redis-rb, em-hiredis, em-redis |
| Scala | scala-redis, redis-client-scala-netty, sedis, scala-redis-client |
| Scheme | redis-client |
| Smalltalk | Smalltalk Redis Client |
| Tcl | Tcl Client |

Tabella 2.1: Librerie con supporto a Redis

Per questa tesi è stato scelto come linguaggio di programmazione Python 2.7.5 la libreria redis-py [6] che con pochissime righe di codice permette subito di operare nella nostra base di dati, ecco un semplice esempio dove viene salvato e stampato una String.

```
import redis
r = redis.Redis(host='localhost', port=6379, db=0)
r.set("name", "value")
print r.get("name")
```

Esempio con pipeline

```
import redis
r = redis.Redis(host='localhost', port=6379, db=0)
r.set('bing', 'baz')
# creo un'istanza di pipeline
pipe = r.pipeline()
# accodo comandi nel buffer
pipe.set('foo', 'bar')
pipe.get('bing')
# eseguo l'intera pipe
pipe.execute()
[True, 'baz']
```

[6] https://github.com/andymccurdy/redis-py

# Capitolo 3

# Progettazione “clone di Twitter”

In questo capitolo, prendendo come spunto lo stesso caso di studio presente sul sito ufficiale di Redis `http://redis.io/topics/twitter-clone`, cercheremo di dare una nostra versione migliorata e più simile al noto servizio di micro-blogging. Oltre alla progettazione del database scriveremo alcune righe di codice in Python per simulare un utilizzo pratico del nostro servizio.

Il nostro Twitter non avrà tutte le funzionalità di quello originale, ma si focalizzerà sulle caratteristiche base del noto servizio. Abbiamo deciso che ogni utente si potrà registrare scegliendo uno username, password ed email. Potrà decidere a propria scelta di seguire (following) altri utente i quali non potranno rifiutarsi di essere seguiti (followers).

Nella pagina principale di ogni utente compariranno in ordine cronologico tutti i tweet scritti da lui e dagli utenti che lui ha deciso di seguire. Nel momento in cui un utente decide di seguire o di smettere di seguire un altro utente la sua time-line deve aggiornarsi e mostrare solo i post interessati. Il numero di tweet è illimitato, ma come sul servizio originale non potranno superare i 160 caratteri.

## 3.1 Progettazione con Redis

Nella progettazione di un database relazionale dobbiamo imbatterci in tabelle, relazioni ed indici, ma in un database NoSql chiave/valore non abbiamo nulla di tutto ciò, quindi con cosa possiamo progettarlo? Abbiamo bisogno di identificare quali chiavi sono necessarie per rappresentare i nostri oggetti e che tipo di valori devono trattare. La rigidità con la quale siamo abituati a progettare le nostre basi di dati tende a scomparire ed abbiamo la possibilità di adattarci alle richieste ed ottenere maggiori performance. Una delle poche cose da stare attenti è quella di scegliere nomi di chiavi non troppo lunghe in modo da evitare uno spreco di memoria inutile.

Ora andiamo ad analizzare ogni oggetto che interessa la nostra base di dati ed assegnamogli un tipo di struttura Redis.

### 3.1.1 Utenti

Per ogni utente del nostro microblog dovremmo tener traccia di: nome utente, ID utente, password, follower e following, e così via.
La prima domanda è, che cosa dovrebbe identificare un utente all’interno del nostro sistema? Il nome utente può essere una buona idea dal momento che è unico, ma è anche troppo grande e non vogliamo rimanere a corto di memoria quindi è meglio se lo identifichiamo con un ID numerico.
La generazione di ID univoci la possiamo ottenere utilizzando un solo valore String

```
nextUserId [String]
```

sul quale eseguiremo il comando INCR che in una operazione atomica ne incrementa il valore e lo ritorna.

Per ogni utente immagazzineremo username, password ed email ecc all'interno di un Hash la cui chiave sarà “uid:<ID>”. Abbiamo scelto un Hash perchè Redis offre questa struttura molto pratica se un'entità ha più di un valore da associarvi. Se non esistesse l'Hash il modo classico di procedere sarebbe stato quello di creare delle coppie chiave/valore diverse per ogni informazione da archiviare oppure salvando tutti i dati codificati ad esempio in JSON all'interno di una String.

```
uid:<ID> [hash]
- usrn
- pwd
- mail
```

Oltre a questo per ogni utente creeremo la chiave

```
usr:<username>:uid [String]
```

che ci permetterà di archiviare l'id di ogni utente conoscendone lo username.

Il problema di come organizzare following e followers è abbastanza intuitivo, difatti faremmo uso di 2 Set differenti

```
uid:<ID>:following [Set]
uid:<ID>:followers [Set]
```

### 3.1.2 Post

I post (o tweet), che ogni utente pubblica, avranno i seguenti 3 campi archiviati dentro ad un Hash.

```
post:<ID> [Hash]
- u
- t
- b
```

In **u** inseriremo l'id dell'utente che ha scritto il tweet, in **t** il timestamp e in **b** il corpo del testo. È importante scegliere chiavi molto corte per evitare spreco di memoria.

Come per gli utenti avremmo bisogno di una String per generare gli id dei post

```
nextPostId [String]
```

Non ci resta che organizzare i tweet e fare in modo che compaiano all'interno delle varie time-line dei followers. Per fare questo utilizzeremo la struttura dati più potente, ma anche la più dispendiosa in termini di memoria occupata, ovvero un Zset. Ogni time-line sarà rappresentata da un Zset ordinato per timestamp che conterrà gli id dei post. Ogni qualvolta un utente scriverà un tweet si dovranno aggiornare tutti i Zset dei suoi followers aggiungendo l'id del post. Si dovrà inoltre ricordarsi di aggiornare il set ordinato di ogni utente ogni qualvolta questi decide di seguire o non seguire più un altro utente, inserendo o rimuovendo tutti gli id dei tweet non più rilevanti.

```
uid:<ID>:tline [Zset]
```

Si sarebbe potuto realizzare una versione un po' più semplice sostituendo i Zset con delle semplici List ed eliminando gli aggiornamenti in fase di aggiunta/rimozione dei followers, ma non si avrebbe avuto una time-line "veritiera" che potesse ricreare la storia cronologica di tutti i post.

```
uid:<ID>:posts [List]
```

## 3.2 Database Twitter con MySql

Abbiamo appena progettato un sistema di microblogging in Redis, ora vogliamo reimplementare il tutto con MySql e quindi riprogettarlo in modo relazionale. Andiamo passo passo a fare un'analisi del progetto fino ad ottenere le query per la realizzazione delle strutture dati su MySql.

### 3.2.1 Raccolta e analisi requisiti

Ogni **utente** registrato è caratterizzato da username, email e una password per accedere al servizio ed è identificato univocamente da un id numerico. Ogni utente può seguire (following) o essere seguito (followers) da un numero indefinito di utenti. Un utente registrato può scrivere dei **post** i quali verranno identificati univocamente da un id numerico e saranno caratterizzati da un testo e data/ora. Per praticità le date saranno archiviate come timestamp quindi in formato numerico double.

### 3.2.2 Progettazione concettuale

Il modello concettuale utilizzato è il modello Entità-Relazione, eccone lo schema.

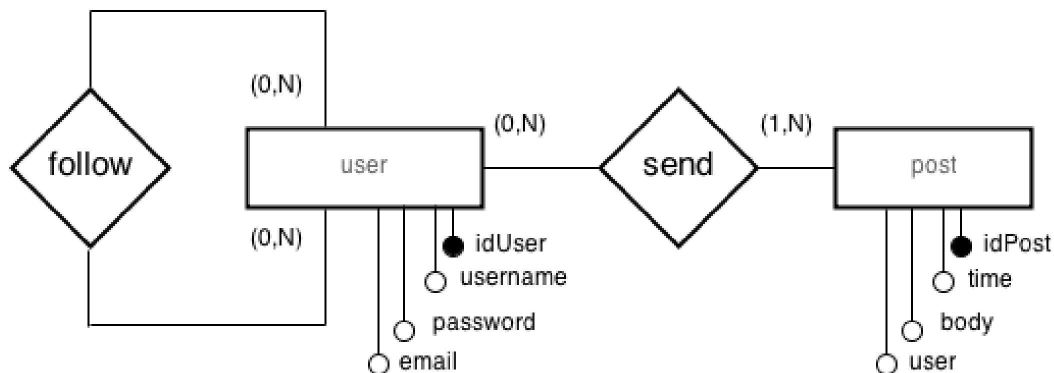


Figura 3.1: Schema ER clone twitter

### 3.2.3 Progettazione logica

Non abbiamo bisogno di nessuna fase di ristrutturazione dello schema ER, possiamo quindi creare lo schema logico equivalente. Il suddetto schema è qui riportato in figura.

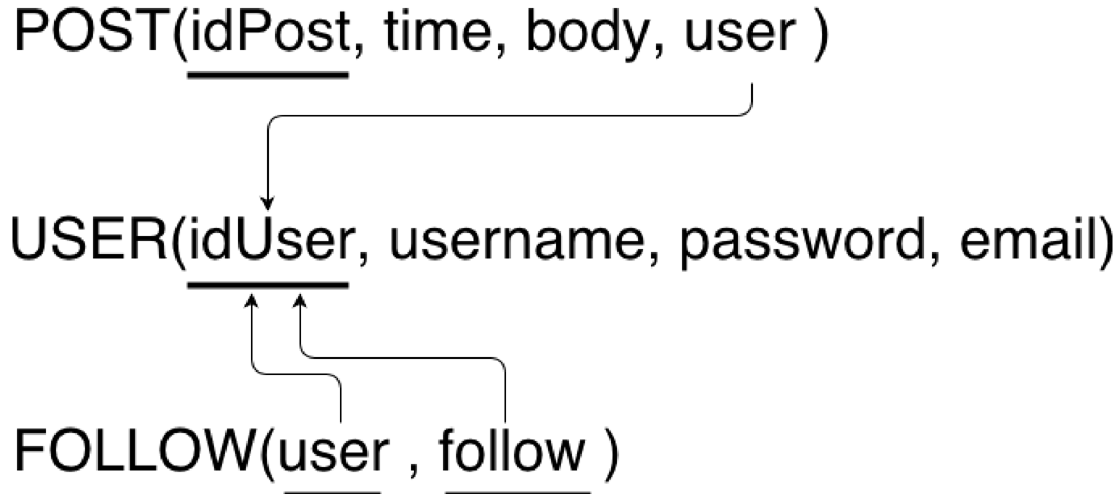


Figura 3.2: Schema Logico clone twitter

### 3.2.4 Progettazione fisica

```
--
-- Struttura della tabella ‘follow‘
--

CREATE TABLE IF NOT EXISTS ‘follow‘ (
  ‘user‘ int(10) unsigned NOT NULL,
  ‘follow‘ int(10) unsigned NOT NULL,
  PRIMARY KEY (‘user‘,‘follow‘),
  KEY ‘follow‘ (‘follow‘)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- --------------------------------------------------------

--
-- Struttura della tabella ‘post‘
--

CREATE TABLE IF NOT EXISTS ‘post‘ (
  ‘idPost‘ int(10) unsigned NOT NULL AUTO_INCREMENT,
  ‘time‘ double NOT NULL,
  ‘body‘ varchar(160) NOT NULL,
  ‘user‘ int(10) unsigned NOT NULL,
  PRIMARY KEY (‘idPost‘),
  KEY ‘user‘ (‘user‘)
```

```
) ENGINE=InnoDB  DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;

-- --------------------------------------------------------

--
-- Struttura della tabella ‘user‘
--

CREATE TABLE IF NOT EXISTS ‘user‘ (
  ‘idUser‘ int(10) unsigned NOT NULL AUTO_INCREMENT,
  ‘username‘ varchar(32) NOT NULL,
  ‘password‘ varchar(64) NOT NULL,
  ‘email‘ varchar(32) NOT NULL,
  PRIMARY KEY (‘idUser‘)
) ENGINE=InnoDB  DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;

--
-- Limiti per le tabelle scaricate
--

--
-- Limiti per la tabella ‘follow‘
--
ALTER TABLE ‘follow‘
  ADD CONSTRAINT ‘follow_ibfk_1‘ FOREIGN KEY (‘user‘) REFERENCES ‘user‘ (‘idUser‘),
  ADD CONSTRAINT ‘follow_ibfk_2‘ FOREIGN KEY (‘follow‘) REFERENCES ‘user‘ (‘idUser‘);

--
-- Limiti per la tabella ‘post‘
--
ALTER TABLE ‘post‘
  ADD CONSTRAINT ‘post_ibfk_1‘ FOREIGN KEY (‘user‘) REFERENCES ‘user‘ (‘idUser‘);
```

# Capitolo 4

# Analisi prestazionale

In questo capitolo confronteremo le prestazioni di MySql e Redis, analizzeremo prima alcune operazioni di inserimento per avere un primo paragone e poi passeremo a query più complesse che interesseranno le due basi di dati progettate al capito precedente.

La macchina utilizzata nei test è un portatile Dell equipaggiato con 2 Gb di Ram, cpu Intel® Core® 2 Duo CPU T5670 @ 1.80GHz il sistema operativo è Debian testing "Jessie" con linux 3.10-2-amd64

## 4.1 Inserimenti

Per quanto riguarda Redis analizzeremo gli inserimenti in tutte e cinque le strutture dati che questo database NoSQL mette a disposizione. Le tuple che andremo ad inserire avranno questi campi:

```
'title': 'article title'
'link': 'http://en.wikipedia.org/wiki/Redis'
'poster': 'user:username'
'time': '1380182882.437284'
'votes': '132'
```

Ogni inserimento avrà una chiave/id numerica, il campo time sarà aggiornato, mentre il campo votes verrà generato random.

Per prendere nota dei tempi di esecuzione leggeremo da Python lo stato dell'orologio del sistema operativo prima e dopo il segmento di codice da analizzare e ne faremo una differenza.

Nei caso in cui non sarà utilizzato alcun linguaggio di programmazione faremmo uso del comando POSIX 'time' che a sua volta esegue un comando e ne ritorna il tempo impiegato per la sua esecuzione.

### 4.1.1 Inserimento in Redis

I metodi di inserimento utilizzati per il Redis sono i seguenti:

1. Inserimento di tuple in Python
2. Inserimento di tuple in Python con pipeline
3. Inserimento tramite file

#### 4.1.1.1 Python

Per questo test abbiamo effettuato l'inserimento di N tupe eseguendo all'interno di un ciclo for l'istruzione per il loro inserimento.

| N | string | hash | list | set | zset |
|---|---|---|---|---|---|
| 100 | 0.0117s | 0.0159s | 0.0127s | 0.0114s | 0.0127s |
| 1 000 | 0.1093s | 0.3101s | 0.1096s | 0.1095s | 0.1238s |
| 10 000 | 1.1364s | 2.4766s | 1.9481s | 1.5588s | 2.5600s |
| 100 000 | 18.167s | 22.9338s | 16.6832s | 18.0397s | 22.0138s |
| 1 000 000 | 2m35.0374s | 3m33.7781s | 2m53.8825s | 3m5.5705s | 3m16.1396s |

Tabella 4.1: Benchmark inserimenti Redis Python

#### 4.1.1.2 Python Pipeline

In questa tabella notiamo i miglioramenti che le pipeline introducono eliminando i tempi di attesa tra l'invio di più comandi uno di seguito all'altro.

| n | string | hash | list | set | zset |
|---|---|---|---|---|---|
| 100 | 0.0050s | 0.0089 s | 0.0436 s | 0.0043s | 0.0052s |
| 1 000 | 0.0357s | 0.0710s | 0.0316s | 0.0317s | 0.0412s |
| 10 000 | 0.3251s | 0.6915s | 0.3046s | 0.3315s | 0.4084s |
| 100 000 | 3.4660s | 7.2618 s | 3.4859s | 3.4265s | 4.4547s |
| 1 000 000 | 36.4845s | 1m20.7549s | 35.6742s | 34.7857s | 44.3483s |

Tabella 4.2: Benchmark inserimenti Redis Python Pipeline

#### 4.1.1.3 File txt

Il file txt è stato generato tramite Bash e successivamente rediretto a redis-cli da terminale utilizzando il comando 'head -n 1000' che accetta come parametro il numero di linee da stampare.

```
$ time head -n 1000 data.txt | redis-cli
```

È stato quindi necessario generare cinque tipi di file, ognuno con le istruzioni riguardanti la corretta struttura dati da analizzare.

| * | string | hash | list | set | zset |
|---|---|---|---|---|---|
| 100 | 0.022s | 0.032s | 0.025s | 0.026s | 0.059s |
| 1 000 | 0.096s | 0.112s | 0.119s | 0.119s | 0.125s |
| 10 000 | 0.829s | 1.021s | 0.941s | 0.916s | 0.990s |
| 100 000 | 8.339s | 10.102s | 9.105s | 9.139s | 9.915s |
| 1 000 000 | 1m25.926s | 1m42.106s | 1m32.089s | 1m32.459s | 1m40.218s |

Tabella 4.3: Benchmark inserimenti Redis file

### 4.1.2 Inserimento in MySql

I metodi di inserimento utilizzati in MySql sono i seguenti:

1. Inserimento di tuple in Python
2. Inserimento in Bulk di tuple in Python (pipeline)
3. Inserimento tramite file

#### 4.1.2.1 Python

Esempio di inserimento di una tupla in MySql tramite prepared statement.

```
con = mdb.connect('localhost', 'utente', 'pwd', 'blog')
query = """
  INSERT INTO test (id, title, link, poster, time, votes)
  VALUES (%s, %s, %s, %s, %s, %s)
  """
with con:
  cur = con.cursor()
  for i in range(n):
    cur.execute(query,(
            str(i+1),
            'article title',
            'http://en.wikipedia.org/wiki/Redis',
            'user:username',
            str(now),
            str(vote)
            ))
```

Con la keyword **with** l'interprete python rilascia in automatico la risorsa eseguendo le query non appena viene invocato il metodo *execute()*.

| * | time |
|---|---|
| 100 | 5.0612s |
| 1 000 | 48.8872s |
| 10 000 | 8m2.3998s |
| 100 000 | 1h19m34.4091s |
| 1 000 000 | 13h13m29.2990s |

Tabella 4.4: Benchmark inserimenti MySql Python

#### 4.1.2.2 Python bulk

Esempio di inserimento tramite l'uso di una pipe, la quale non viene rilasciata in automatico ma solo nel momento in cui viene invocato il metodo **commit()**.

```
con = mdb.connect('localhost', 'utente', 'pwd', 'blog')
query = """
  INSERT INTO test (id, title, link, poster, time, votes)
  VALUES (%s, %s, %s, %s, %s, %s)
  """
cur = con.cursor()
for i in range(n):
  cur.execute(query,(
            str(i+1),
            'article title',
            'http://en.wikipedia.org/wiki/Redis',
            'user:username',
            str(now),
            str(vote)
            ))
con.commit()
```

| * | time |
|---|---|
| 100 | 0.0666s |
| 1 000 | 0.2112s |
| 10 000 | 1.7690s |
| 100 000 | 18.4094s |
| 1 000 000 | 3m7.4183s |

Tabella 4.5: Benchmark inserimenti MySql Python Pipeline

#### 4.1.2.3 txt

Questo test è stato eseguito in due diverse modalità, la prima creando file contenenti singoli INSERT, la seconda scrivendo sempre su file le query INSERT in bulk da 500 record. I file li abbiamo poi caricati in MySql tramite Bash

```
time mysql -u utente -p blog < dump.sql
```

Query singole

```
...
1) INSERT INTO ‘test‘ (‘id‘, ‘title‘, ‘link‘, ‘poster‘, ‘time‘, ‘votes‘) VALUES(97, ’article title’,
   ’http://en.wikipedia.org/wiki/Redis’, ’user:username’, 1380197107, 283);
2) INSERT INTO ‘test‘ (‘id‘, ‘title‘, ‘link‘, ‘poster‘, ‘time‘, ‘votes‘) VALUES(98, ’article title’,
   ’http://en.wikipedia.org/wiki/Redis’, ’user:username’, 1380197107, 205);
3) INSERT INTO ‘test‘ (‘id‘, ‘title‘, ‘link‘, ‘poster‘, ‘time‘, ‘votes‘) VALUES(99, ’article title’,
   ’http://en.wikipedia.org/wiki/Redis’, ’user:username’, 1380197107, 27);
...
```

| * | time |
|---|---|
| 100 | 7.169s |
| 1 000 | 52.958s |
| 10 000 | 8m24.442s |
| 100 000 | 1h19m34.324s |
| 1 000 000 | 13h52m32.625 |

Tabella 4.6: Benchmark inserimenti singoli MySql File

Query in bulk

```
INSERT INTO ‘test‘ (‘id‘, ‘title‘, ‘link‘, ‘poster‘, ‘time‘, ‘votes‘) VALUES
(1, ’article title’, ’http://en.wikipedia.org/wiki/Redis’, ’user:username’, 1384101035, 134),
(2, ’article title’, ’http://en.wikipedia.org/wiki/Redis’, ’user:username’, 1384101035, 19),
(3, ’article title’, ’http://en.wikipedia.org/wiki/Redis’, ’user:username’, 1384101035, 43),
(4, ’article title’, ’http://en.wikipedia.org/wiki/Redis’, ’user:username’, 1384101035, 343),
...
```

| * | time |
|---|---|
| 100 | 0.853s |
| 1 000 | 1.701s |
| 10 000 | 2.859s |
| 100 000 | 16.387s |
| 1 000 000 | 2m24.701s |

Tabella 4.7: Benchmark inserimenti bulk 500 MySql File

### 4.1.3 Analisi

Notiamo un netto miglioramento delle prestazioni quando vengono utilizzate le pipeline, sia in Redis che in MySql; riguardo l’inserimento tramite file ha portato un miglioramento considerevole in Redis mentre in MySql sembra allinearsi alle tempistiche registrate con python.

Di tutti questi dati riportiamo come unico grafico quello riguardante gli inserimenti tramite pipe che riesce a mettere in evidenza sia le differenze tra i due database che tra le cinque strutture dati di Redis mostrando come Hash e Zset, offrendo caratteristiche molto interessanti, siano però meno performanti.

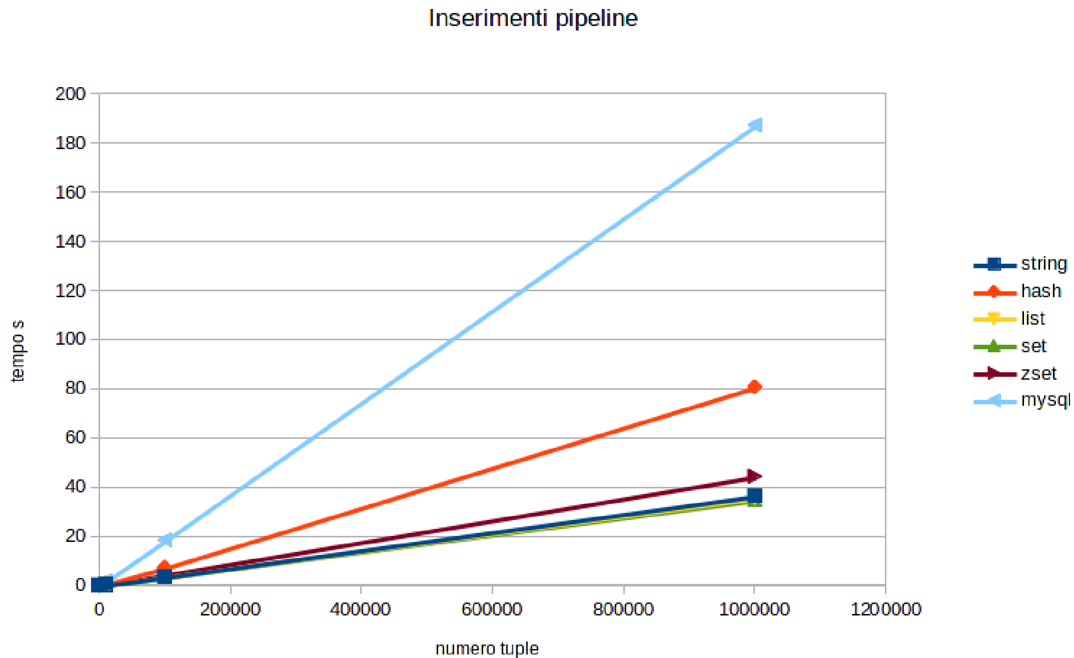


Figura 4.1: Grafico inserimenti pipeline

## 4.2 Query

Analizziamo il comportamento delle 2 differenti implementazioni del nostro clone di Twitter. Inizieremo simulandone l'utilizzo da parte di un ristretto set di utenti, aumentandone man mano il numero e annotando tempistiche e quantità di memoria utilizzata.

I test verranno eseguiti simulando 10,50,100,200,500 e 1000 utenti. Per considerare il caso peggiore ogni utente sarà follower di tutti gli altri ed ognuno scriverà 10 tweet a testa.

I tempi di cui prenderemo nota saranno quelli necessari a generare, per ogni singolo utente, la time-line che dovrebbe visualizzare nella pagina principale del suo account.

Per quanto riguarda Redis questo è il codice della funzione python che genera la bacheca di un utente passando come parametro il suo ID

```
def show_timeline(myid):
  result = []
  for idPost in r.zrevrange("uid:%d:tline" % myid, 0, 9):
    tmp = r.hmget("posts:%s" % idPost, ["u", "t", "b"])
    tmp[0] = r.hget("uid:%s" % tmp[0], "usrn")
    result.append(tmp)
```

per MySql invece il codice è il seguente

```
def show_timeline(myid):
  cur = con.cursor()
  sql = """
  SELECT user.username,body,time FROM user AS u
```

```
  INNER JOIN follow ON u.idUser = follow.user
  INNER JOIN user ON follow.follow = user.idUser
  INNER JOIN post ON follow.follow = post.user
  WHERE u.idUser = %s

  UNION

  SELECT user.username,body,time FROM post
  INNER JOIN user ON user.idUser = post.user
  WHERE post.user = %s

  ORDER BY time DESC
  LIMIT 10
  """
  cur.execute(sql, (myid, myid))
  con.commit()
  cur.close()
```

I tempi di esecuzione verranno calcolati molto semplicemente da codice python

```
starttime = time.time()
for i in range(num_user):
  show_timeline(i)
endtime = time.time()
elapsed = endtime - starttime
print elapsed
```

questi i risultati che sono emersi

| utenti | MySql | Redis |
|---|---|---|
| 10 | 0,4771s | 0,0229s |
| 50 | 0,6020s | 0,1241s |
| 100 | 0,9493s | 0,3289s |
| 200 | 2,2595s | 0,7192s |
| 500 | 11,9209s | 1,3887s |
| 1000 | 46,4772s | 2,8714s |

Tabella 4.8: Confronto Redis Vs MySql query costruzione time-line

Inserendo questi dati in un grafico notiamo subito l'andamento lineare di Redis e un comportamento esponenziale di MySql. Nel prestazioni che abbiamo ottenuto con questo NoSQL key/-value sono possibili solo grazie alla forte ridondanza delle informazioni archiviate, permettendo così di scalare molto meglio rispetto alle classiche soluzioni relazionali.

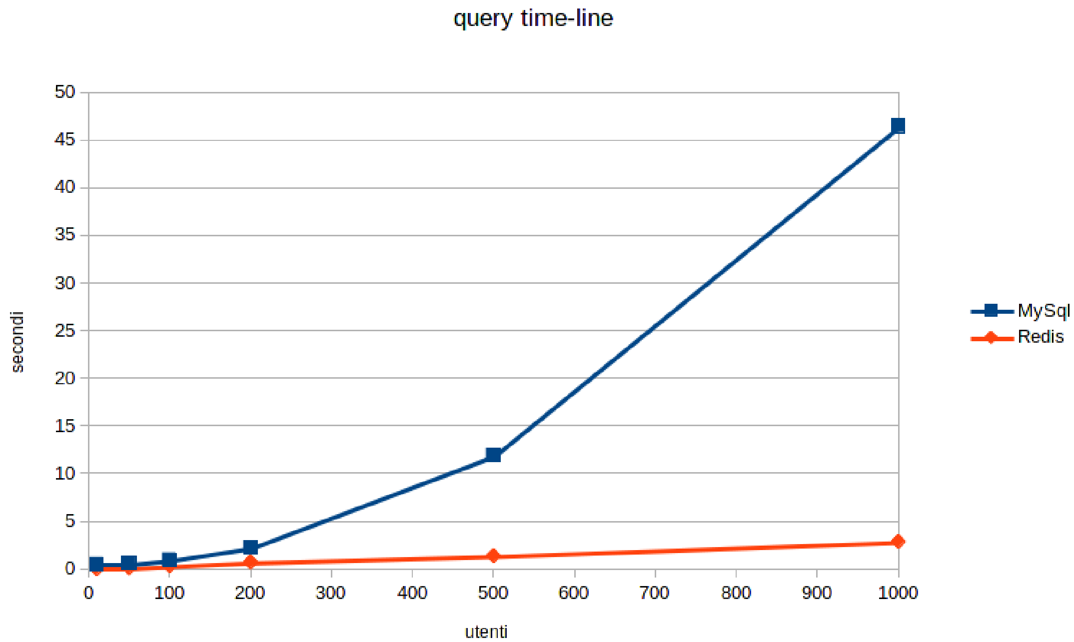


Figura 4.2: Grafico prestazioni query Redis Vs MySql

Possiamo migliorare le prestazioni trasformando la funzione python in uno script Lua, facendolo valutare all'interprete integrato in Redis con un'unica invocazione del comando EVAL, e riducendo al minimo i tempi di esecuzione.

```
def show_timeline(myid):
  lua = """
  local result = {}
  local items = redis.call('zrevrange', KEYS[1], 0, 9)

  for i, postId in ipairs(items) do
    local tmp = {}

    local userId = redis.call('HGET', 'posts:' .. postId, 'u')
    local username = redis.call('HGET', 'uid:' .. userId , 'usrn')
    table.insert(tmp, username)

    local post = redis.call('HMGET', 'posts:' .. postId, 't', 'b')
    for k, v in ipairs(post) do
      table.insert(tmp, v)
    end

    table.insert(result, tmp)
  end

  return result

  """
```

```
return r.eval(lua, 1, "uid:%d:tline" % myid)
```

Dai dati notiamo un incremento delle prestazioni davvero significativo ( 500%)

| utenti | MySql | Redis | Redis LUA |
|---|---|---|---|
| 10 | 0,4771s | 0,0229s | 0,0054s |
| 50 | 0,6020s | 0,1241s | 0,0248s |
| 100 | 0,9493s | 0,3289s | 0,0475s |
| 200 | 2,2595s | 0,7192s | 0,1096s |
| 500 | 11,9209s | 1,3887s | 0,2366s |
| 1000 | 46,4772s | 2,8714s | 0,4717s |

Tabella 4.9: Benchmark query Redis con script Lua costruzione time-line

Grazie al comando EVAL abbiamo eliminato tutti i ritardi introdotti dal socket che deve gestire molte richieste esterne ad una pipe; se il link fosse stato, anziché sul device di loopback su una rete lan o addirittura Internet, le differenze sarebbero state ancora più evidenti. Non riportiamo il grafico perchè visivamente è molto simile al precedente.

Analizziamo ora la quantità di memoria utilizzata dai 2 database riportando i valori in MByte su questa tabella e mostrandoli in un grafico.

| utenti | MySql | Redis |
|---|---|---|
| 10 | 0,08MB | 0,62MB |
| 50 | 4,19MB | 3,00MB |
| 100 | 11,21MB | 9,89MB |
| 200 | 17,45MB | 37,12MB |
| 500 | 31,94MB | 251,44MB |
| 1000 | 49,91MB | 1044,48MB |

Tabella 4.10: Confronto utilizzo memoria Redis Vs MySql

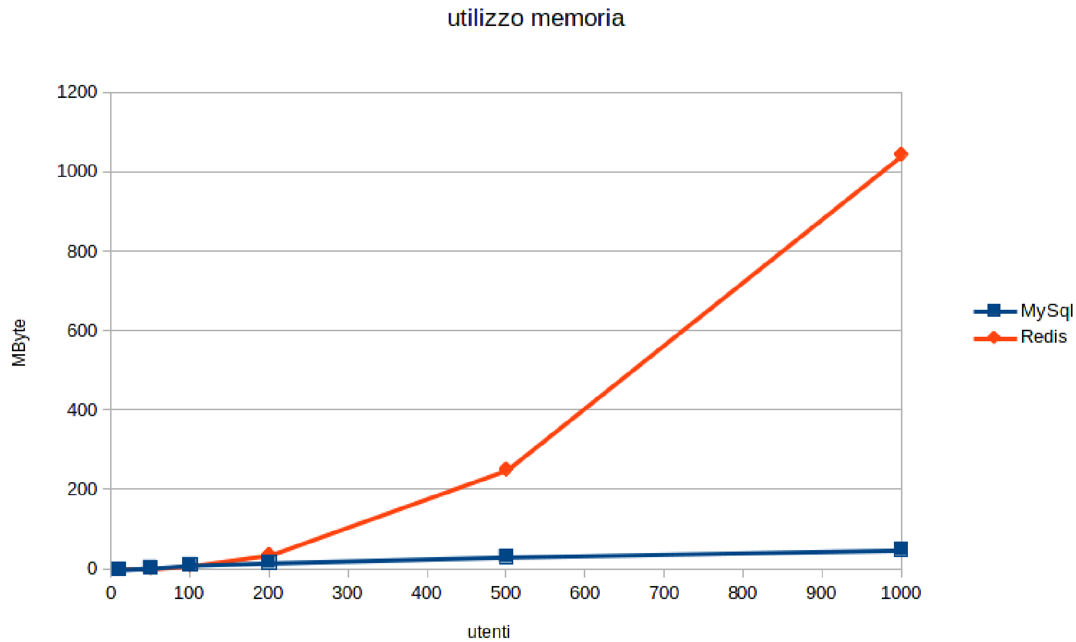


Figura 4.3: Grafico utilizzo memoria Redis Vs MySql

Notiamo l'enorme consumo di memoria da parte di Redis, dovuto prevalentemente dai Zset che mantengono ordinate le time-line di ogni singolo utente.

Dai risultati ottenuti non possiamo certo garantire una facile scalabilità dato l'enorme consumo di memoria (RAM) che Redis necessita per questa specifica realizzazione; possiamo provare a semplificare il nostro progetto alleggerendo la ridondanza dei dati.

## 4.3 Ristrutturazione database Redis

Le strutture che appesantiscono il nostro database NoSQL sono i vari Zset che riempiono esponenzialmente la memoria ram all'aumentare degli utenti. La prima soluzione proposta è quella di provare a rimuoverli tutti ed a sostituirli con uno solo che manterrà ordinati tutti i tweet di tutti gli utenti.

`posts:tline [Zset]`

Non ci resta che modificare la nostra query in Lua e confrontare i dati con le tempistiche precedenti. Per ogni utente di cui vogliamo costruire la time-line, lo script accederà sequenzialmente ai post ordinati all'interno del Zser posts:tline (partendo dai post più recenti), valuterà se il proprietario appartiene all'insieme dei follower dell'utente e in caso affermativo andrà a pescarsi tutti i dati di quel post, ripeterà queste operazioni fintanto che non avrà collezionato 10 tweet diversi.

```
def show_timeline(myid):
  lua = """
  local result = {}
```

```
local count = 0
local i = 0
while count < 10 do
  local item = redis.call('zrevrange', KEYS[1], i, i)
  local postId = item[1]
  local userId = redis.call('HGET', 'posts:' .. postId, 'u')

  if redis.call('SISMEMBER', KEYS[2], userId) == 1 then

    local tmp = {}

    local username = redis.call('HGET', 'uid:' .. userId , 'usrn')
    table.insert(tmp, username)

    local post = redis.call('HMGET', 'posts:' .. postId, 't', 'b')
    for k, v in ipairs(post) do
      table.insert(tmp, v)
    end

    table.insert(result, tmp)

    count = count + 1
  end
  i = i + 1
end

return result

"""
return r.eval(lua, 2, "posts:tline", "uid:%d:following" % myid, myid)
```

Tempi esecuzione query

| utenti | MySql | Redis LUA | Redis LUA 1Zset |
|---|---|---|---|
| 10 | 0,4771s | 0,0054s | 0,0069s |
| 50 | 0,6020 s | 0,0248s | 0,0306s |
| 100 | 0,9493 s | 0,0475s | 0,0592s |
| 200 | 2,2595 s | 0,1096s | 0,1220s |
| 500 | 11,920 s | 0,2366s | 0,2955s |
| 1000 | 46,4772s | 0,4717s | 0,6510s |

Tabella 4.11: Benchmark Redis reimplementazione con Zset

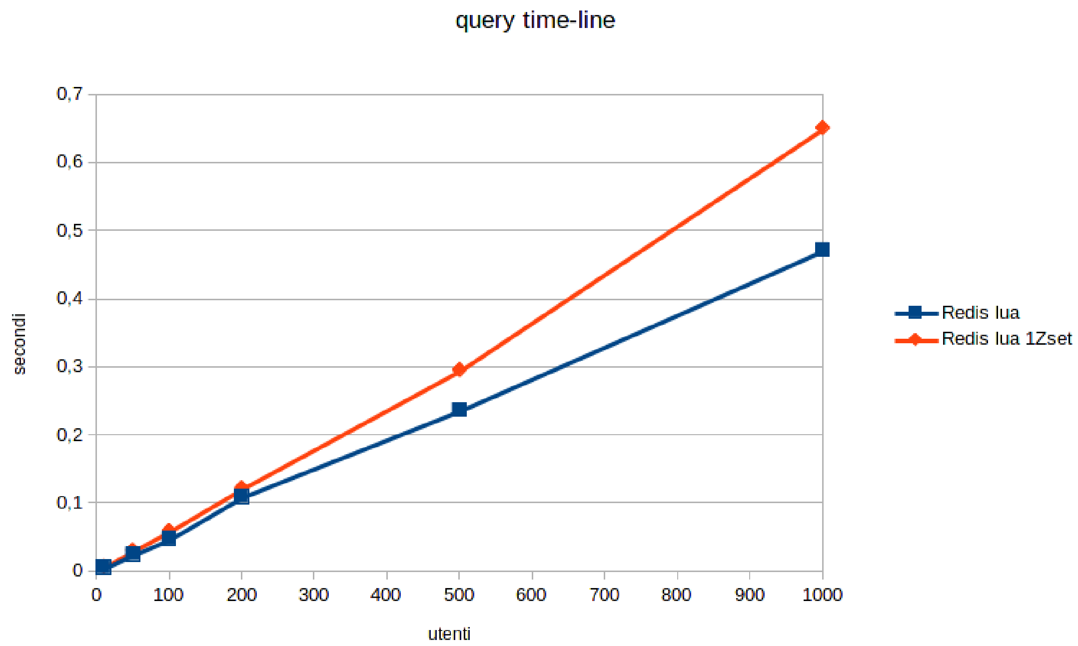


Figura 4.4: Grafico prestazioni query Redis Lua Vs Redis Lua 1Zset

Utilizzo memoria

| utenti | MySql | Redis | New Redis |
|---|---|---|---|
| 10 | 0,08MB | 0,62MB | 0,54MB |
| 50 | 4,19MB | 3,00MB | 0,66MB |
| 100 | 11,21MB | 9,89MB | 0,82MB |
| 200 | 17,45MB | 37,12MB | 1,21MB |
| 500 | 31,94MB | 251,44 MB | 2,92MB |
| 1000 | 49,91MB | 1044,48MB | 39,48MB |

Tabella 4.12: Utilizzo memoria Redis reimplementazione con Zset

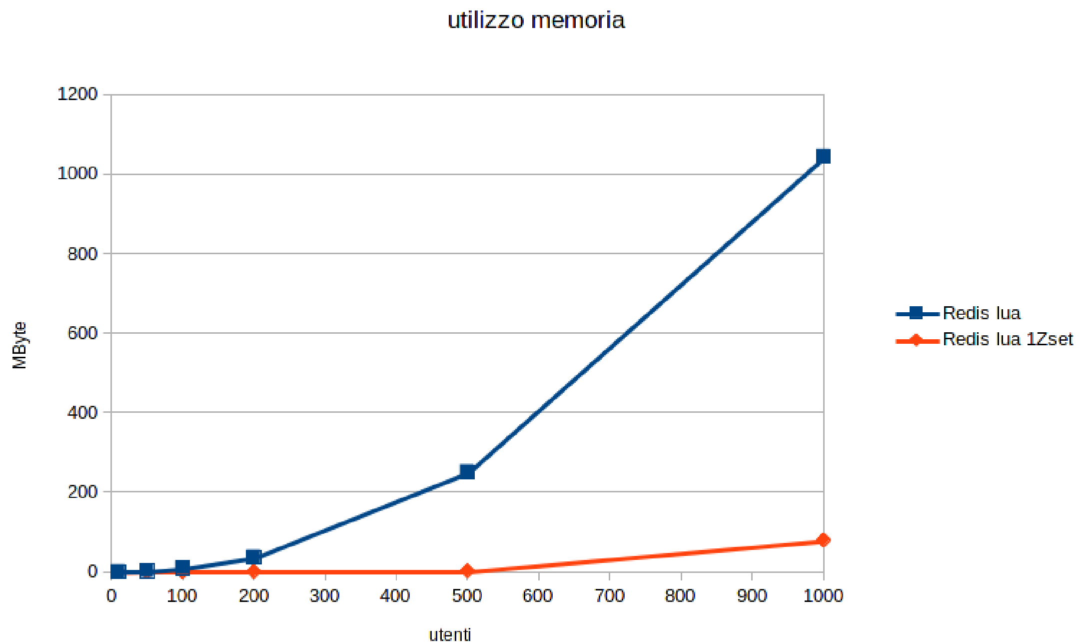


Figura 4.5: Grafico utilizzo memoria Redis Lua Vs Redis Lua 1Zset

Notiamo che pur mantenendo prestazioni molto simili alla prima realizzazione (riduzione prestazioni  20%) siamo riusciti a rendere il problema della memoria un fattore secondario.

Un'altra possibile soluzione è quella di utilizzare al posto di uno Zset una List, inserendo i tweet in testa e gestendola come una pila. Le modifiche sono minime e i risultati sono molto simili a quelli ottenuti con un solo Zset.

Nella precedente simulazione abbiamo considerato ogni utente follower di tutti gli altri, ciò implica che le time-line, oltre ad essere composte tutte dagli stessi tweet, pescano sempre i primi 10 post in testa allo Zset o alla List: si tratta del caso migliore. Abbiamo quindi effettuato lo stesso test (query di generazione delle time-line) cercando di randomizzare alcuni fattori ed allineandoci al caso medio.

1. numero fisso di follower (10) per ogni utente e un numero medio (random) di 10 post per utente.

2. numero random di follower per ogni utente e un numero medio (random) di 10 post per utente.

Sono emersi i seguenti risultati

Primo test con "pochi" followers

| utenti | Zset | List | MySql |
|---|---|---|---|
| 100 | 0,2283s | 0,1996s | 0,1207s |
| 200 | 0,8037s | 0,7029s | 0,2386s |
| 500 | 4,8368s | 4,4681s | 0,6357s |
| 750 | 11,2475s | 10,5336s | 1,1332s |
| 1000 | 19,6255s | 19,6856s | 1,4536s |
| 2000 | 1m18,3133s | 1m34,2718s | 2,6406s |

Tabella 4.13: Benchmark Redis Zset - Redis List - MySql query con "pochi" followers

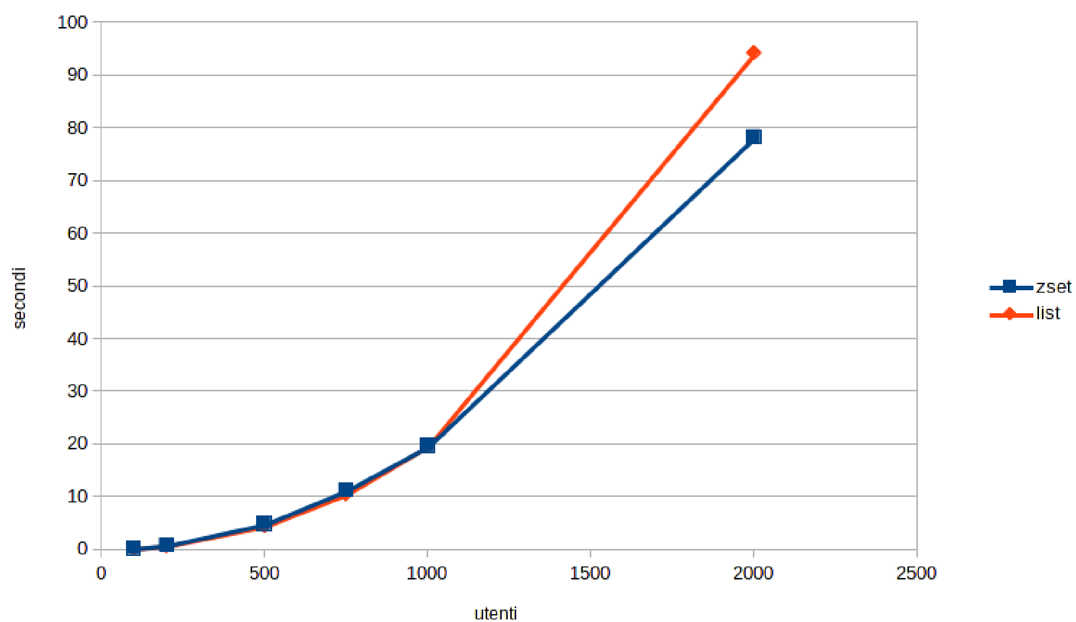


Figura 4.6: Grafico prestazioni query Redis con 10 followers

Secondo test con "tanti" followers

| utenti | Zset | List | MySql |
|---|---|---|---|
| 100 | 0,0898s | 0,0812s | 0,4579s |
| 200 | 0,1971s | 0,1795s | 1,2911s |
| 500 | 0,5975s | 0,5426s | 7,3026s |
| 750 | 0,9188s | 0,8459s | 15,6507s |
| 1000 | 1,2734s | 1,1876s | 30,5189s |
| 2000 | 2,8786s | 2,7015s | 2m8,9656s |
| 3000 | 4,7030s | 4,5912s | 4m46,5334s |
| 4000 | 6,4050s | 6,5662s | 8m30,7964s |

Tabella 4.14: Benchmark Redis Zset - Redis List - MySql query con “tanti” followers

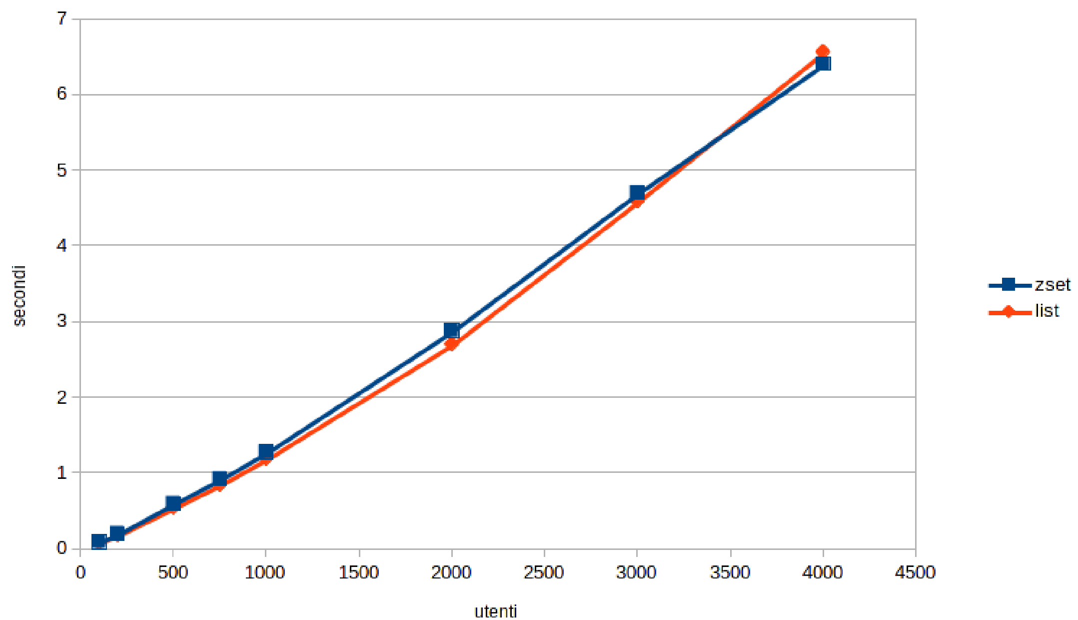


Figura 4.7: Grafico prestazioni query Redis con numero di followers random

Notiamo subito che nel test con “pochi” follower MySql si comporta decisamente meglio rispetto a Redis all’aumentare del numero di utenti presenti nel database, mentre nel test con “tanti” follower la situazione quasi si ribalta.

La scalabilità del database NoSQL viene evidenziata se paragoniamo i dati appena raccolti con queste altre 2 tabelle dove è stato rieseguito il test precedente aumentando i post medi per utente a 100 e successivamente a 200.

100 post medi

| utenti | Zset | List | MySql |
|---|---|---|---|
| 100 | 0,0910s | 0,0939s | 3,9362s |
| 200 | 0,2008s | 0,1869s | 12,3349s |
| 500 | 0,5676s | 0,5146s | 1m22,0576s |
| 750 | 0,9570s | 0,8618s | 3m10,8802s |
| 1000 | 1,3438s | 1,2316s | 12m41,3457s |
| 2000 | 2,8882s | 3,0777s | 50m44,7423s |

Tabella 4.15: Benchmark Redis Zset - Redis List - MySql query con "tanti" followers e 100 post/utente

200 post medi

| utenti | Zset | List | MySql |
|---|---|---|---|
| 100 | 0,0906s | 0,0818s | 7,7892s |
| 200 | 0,2009s | 0,1858s | 27,7168s |
| 500 | 0,6187s | 0,5575s | 7m40,6193s |
| 750 | 0,9269s | 0,8404s | 21m48,7950s |
| 1000 | 1,2449s | 1,1080s | 42m15,9799s |
| 2000 | 2,9868s | 2,8389s | 2h54m23,0803s |

Tabella 4.16: Benchmark Redis Zset - Redis List - MySql query con "tanti" followers e 200 post/utente

Possiamo aggiungere che la soluzione con Zset è preferibile se abbiamo molte query (costruzione delle time-line) e pochi inserimenti di tweet mentre la List nel caso contrario. Questo perchè nello Zset letture e scritture sono entrambe O(log(n)) dove n è il numero di elementi dell'insieme ordinato, mentre la List ha prestazioni O(1) nell'inserimento in testa e O(m) per all'accesso all'm-esimo elemento.

## Capitolo 5

# Conclusioni

In questa tesi, dopo una breve panoramica del mondo dei database NoSql, si è voluto studiare il DBMS NoSql Redis: un particolare database key/value alquanto particolare, primo perché il valore di ogni chiave può essere una delle cinque strutture dati descritte nella sezione 2.7, secondo perché il suo intero dataset è sempre presente in Ram e questo porta ad avere prestazioni in lettura e scrittura molto superiori rispetto alla media degli altri database relazionali e non, i cui dati sono solitamente archiviati solo su disco.
Abbiamo analizzato i due metodi per la persistenza dei dati: RDB (Redis DataBase) e AOF (Append Only File) che riescono a garantire una sincronizzazione dei dati su disco oltre a risultare un facile modo di effettuare backup e recovery dei dati.
Abbiamo poi analizzato, sul piano della sicurezza, la gestione dei permessi, del sistema di accesso e notato l'inesistenza di "NoSQL-injection" dovuta al rigido protocollo che accetta solo comandi di lunghezza prestabilita e parametri prefissati.
Siamo passati alla gestione delle transazioni ed abbiamo visto come viene gestita la replicazione del database su più nodi client-server, anche se, non offrendo nessuna funzione di Map-Reduce, la distribuzione del carico di lavoro sui nodi rimane compito del programmatore.
Dopo aver dato una possibile implementazione di un "clone di Twitter", sia con Redis che con MySql, siamo passati all'effettivo confronto dei due DBMS prima effettuando dei semplici inserimenti e successivamente query più "pesanti" sulla base di dati progettata in precedenza. Con i primi test si è potuto notare le ottime performance di Redis, dovute principalmente al suo utilizzo della Ram, mentre con le query abbiamo mostrato gli effettivi miglioramenti prestazionali che si ottengono grazie alla natura scheme-free di Redis che toglie ogni vincolo riguardante la classica progettazione relazionale, invogliando l'introduzione di ridondanza dei dati, svincolandoci dalla necessità dei JOIN sql e permettendo un'ottima scalabilità. Questa libertà di implementazione ha però evidenziato problemi riguardanti l'eccessivo utilizzo di memoria del database NoSQL rischiando di saturare tutta la memoria messa a disposizione del pc utilizzato nei test (2GB), problema da non sottovalutare sul piano economico. Si è quindi ristrutturato il sistema non ottenendo più prestazioni lineari come al caso precedente, ma risultati comunque molto buoni.
Abbiamo però notato, con altri tipi di test, che queste modifiche sono quasi lineari a carichi di lavoro molto alti, ma risultano molto inefficienti a carichi leggeri quando ad esempio MySql sembra rispondere molto più prontamente. Lo scenario più comune nel mondo NoSQL è di fatti quello di affiancare enormi basi di dati relazionali per migliorare le prestazioni in situazioni particolari; sono un chiaro esempio Facebook e Twitter che per ottimizzare le funzioni di ricerca hanno adottato appunto queste tecnologie.

# Bibliografia


[1] Josiah L. Carlson, *Redis in Action*, Manning Publications, Giugno 2013

[2] Tiago Macedo, Fred Oliveira, *Redis Cookbook*, O'Reilly Media, Luglio 2011

[3] Eric Redmond, Jim R. Wilson, *Seven Databases in Seven Weeks*, Pragmatic Bookshelf, Maggio 2012

[4] Elmasri Ramez A., Navathe Shamkant B. *Sistemi di basi di dati. Fondamenti*, Pearson, 2011

[5] Karl Seguin, *The Little Redis Book*, Gennaio 2012, `http://openmymind.net/redis.pdf`

[6] `http://redis.io/documentation`

[7] `http://redis.io/commands`

[8] `http://nosql-database.org/`

[9] `http://mysql-python.sourceforge.net/MySQLdb.html`

[10] `http://blog.nahurst.com/visual-guide-to-nosql-systems`

[11] `http://zetcode.com/db/mysqlpython/`

[12] `http://liamkaufman.com/blog/2012/06/04/redis-and-relational-data/`